High Availability Architectures Using Pacemaker with KVM on IBM z Systems Servers

Version 1.01
March 21, 2017

Scott Loveland
KVM for IBM z Systems Solution Test
### Table of Content

Abstract .............................................................................................................................................................. 3  
Acknowledgements .................................................................................................................................................. 3  
Introduction: Definition of High Availability ........................................................................................................ 4  
Introduction to High Availability with KVM and LPARs .................................................................................................. 5  
Chapter 1: Scenarios ........................................................................................................................................ 11  
Chapter 2: Reference Architecture: Active-Active KVM Cluster with Opaque VMs ........................................ 12  
  Scenario Being Solved ......................................................................................................................................... 12  
  Architecture Principles ......................................................................................................................................... 12  
  Reference Architecture ........................................................................................................................................ 13  
  Key problem areas ............................................................................................................................................... 18  
  Implementation Notes .......................................................................................................................................... 18  
Chapter 3: Reference Architecture: Active-Active KVM Cluster with Guest Node VMs .................................. 20  
  Scenario Being Solved ......................................................................................................................................... 20  
  Architecture Principles ......................................................................................................................................... 20  
  Reference Architecture ........................................................................................................................................ 21  
  Key problem areas ............................................................................................................................................... 27  
  Implementation Notes .......................................................................................................................................... 27  
Appendix 1: References ................................................................................................................................... 28
Abstract
This paper leverages the experiences of the KVM for IBM z Systems™ solution test team to produce a set of reference architectures that make use of Pacemaker/Corosync to provide high availability clustering for applications running on KVM on IBM z Systems servers. This paper focuses on those architectures that cover the following scenarios:

- Where the application runs on Linux virtual servers under KVM for IBM z.
- Of highest interest to our customers.
- Have not been documented before on System z®.

This paper does not cover:

- All the details necessary to implement the reference architectures (though some implementation notes are provided).
- HA networking considerations. We cover the major components and flow between them. We have not covered how to create a highly available network.
- How to HA-enable your storage subsystem.
- How to define a KVM for IBM z Systems Hypervisor Performance Manager policy and configuration.

Note that while the distribution of KVM used during testing for this paper was KVM for IBM z Systems, the described architectures are applicable to any distribution of KVM running on IBM z Systems® servers.

Acknowledgements
Thanks to these people whose work made this paper possible: Scott Greenlese, Si Bo Niu.
Introduction: Definition of High Availability

For the purposes of this paper, we have adopted the definition used by the IBM® HA Center of Competence in Poughkeepsie, NY.

- **High Availability** – Designed to provide service during defined periods, at acceptable or agreed upon levels, and masks unplanned outages from end-users. It employs Fault Tolerance; Automated Failure Detection, Recovery, Bypass Reconfiguration, Testing, Problem and Change Management

- **Continuous Operations (CO)** – Designed to continuously operate and mask planned outages from end-users. It employs non-disruptive hardware and software changes, non-disruptive configuration, and software coexistence.

- **Continuous Availability (CA)** – Designed to deliver non-disruptive service to the end user seven days a week, 24 hours a day (there are no planned or unplanned outages).

The architectures in this paper strive to provide High Availability. They leverage an external cluster manager to detect failed servers and services, and trigger failover processing. However, delays in the automated recovery of some system components can be long enough to cause user transactions to fail and have to be re-entered. By combining this external cluster approach for server failover with cluster-aware applications (such as IBM’s Websphere Application Server) that leverage internal insight of service processing to dynamically redirect work from failed servers to surviving ones, even higher levels of availability can be achieved.
Introduction to High Availability with KVM and LPARs

KVM for IBM z Systems is always running in a logical partition (LPAR). So we have introduced two new layers between Linux and the hardware, namely KVM and LPAR. These layers play prominently in the availability of your applications, because they provide services that the Linux systems use.

Where are the Single Points of Failure (SPoFs)?

Consider an example where an IBM LinuxONE™ or IBM z Systems® server has one LPAR running KVM to host Linux guests. Your application is installed on a single Linux server (guest). Where are the points of failure? There are several:

- The LinuxONE server hardware could experience multiple unrecoverable failures, causing the entire server to fail.
- The disk subsystem could fail. Note that this paper does not include any information on HA-enabling the disk subsystem.
- The LPAR microcode could fail.
- KVM could fail.
- Linux could fail.
- The application could fail.
- The network connecting the application to its users could fail.

The odds of each failure are different. In this case, the probability of an application failure is highest, while the probability of the LinuxONE server failure is lowest. The others fall on a continuum between those extremes.

So how do we eliminate these single points of failure? An easy and effective method is to eliminate them by duplicating them. Duplicating the application is usually easy, while duplicating the LinuxONE server can be expensive, with the cost and difficulty of the others falling on a continuum between these extremes.

The following table summarizes these points:

<table>
<thead>
<tr>
<th>Single Point of Failure</th>
<th>Probability of Failure</th>
<th>Cost to fix SPoF</th>
</tr>
</thead>
<tbody>
<tr>
<td>LinuxONE server</td>
<td>Very Low</td>
<td>High</td>
</tr>
<tr>
<td>Disk Subsystem</td>
<td>Very Low</td>
<td>Medium</td>
</tr>
<tr>
<td>LPAR</td>
<td>Very Low</td>
<td>Low</td>
</tr>
<tr>
<td>KVM</td>
<td>Low</td>
<td>Low</td>
</tr>
<tr>
<td>Linux</td>
<td>Low</td>
<td>Very Low</td>
</tr>
<tr>
<td>Application</td>
<td>High</td>
<td>Very Low</td>
</tr>
</tbody>
</table>
Besides hardware and software failures, various planned activities can also cause downtime for the application, such as:

- IBM LinuxONE or IBM z Systems hardware upgrades requiring Power On Reset (POR). Note that on current servers, microcode upgrades do not require a POR.
- LPAR configuration changes requiring reboot of the LPAR
- KVM maintenance
- Linux kernel maintenance that requires reboot
- Qemu maintenance that requires reboot of a virtual server
- Application maintenance

There are no probabilities that can be assigned to these since they are directly under the control of the customer. The customer’s policies will dictate how often these will occur.

In order of increasing availability, the following examples examine some possible architectures and their single points of failure.

**Example 1: High Availability not needed**

In this example, an application consisting of three components (an HTTP server, an application server, and a database server) is installed with each component on a single Linux server that runs under KVM. The SPoFs for the application are:

- LinuxONE server
- LPAR
- KVM
- Linux
- Application

Each small box represents a virtual Linux server running as a guest of KVM in a single KVM LPAR. The most likely to fail is some component of the application. When this happens, that application component can be restarted and recovery time will be a few minutes. Or perhaps the entire Linux virtual machine has to be
rebooted and recover time could be even longer. If this recovery time is sufficient, then there is nothing more that needs to be done. If not, then higher availability is needed.

**Example 2: Moderate Availability Needed**

In this example, redundant copies of the application are installed on multiple Linux servers that run under KVM. The SPoFs for the application are reduced to:

- LinuxONE server
- LPAR
- KVM

Each small box represents a virtual Linux server running as a guest of KVM, in a single KVM LPAR.

By simply replicating the application across two or more Linux servers, we have removed the most likely points of failure. Workload is distributed to the duplicated servers so that a failure in any server still leaves another server available to process the workload. Some automatic method is implemented to detect the failure and initiate recovery. When the failed server instance is restarted, workload then begins to flow back to the restarted server, reestablishing full redundancy.

A failure in any Linux server still allows the application to run with the full resources available to it before on the remaining virtual servers. Because all of the CPU and memory is shared among the Linux servers under KVM for IBM z Systems, a failure of one Linux server or application frees up its memory and CPU for use by other Linux servers. For example, if the two application servers are both 80% CPU utilized (using 80% of one CPU), then if one of them fails the other can increase its CPU utilization to 80% of two CPUs. This allows the remaining server to process the full workload immediately, without having to reconfigure the server.
This is very different from failover scenarios in bare metal architectures, where each server must be sized to handle significantly more than its own workload so that the server can have the capacity to handle additional workload if another server fails.

**Example 3: High Availability Needed**

In this example the application is installed on multiple Linux servers that run under multiple KVM systems on multiple LPARs. The SPoFs for the application are reduced to:

- LinuxONE hardware

We have eliminated most of the SPoFs by creating a second LPAR that will run KVM and Linux guests. The application is installed on Linux servers in each LPAR. The cost for doing this is still low, since both LPARs will share the same CPUs, and the real memory can be split between the two LPARs.

A failure in the application, the Linux server, KVM, or LPAR still allows the application to run on the remaining virtual servers with the full resources available to it before the failure. Should one server, KVM, or LPAR fail, the other LPAR can use all of the CPUs that were being shared by both.

Because you are running the same software on the same number of CPUs, software costs do not increase. For all these reasons, this is one of the most cost-effective high availability architectures for KVM for IBM z Systems.

**Example 4: Continuous Availability Needed**
In this example, the application is installed on multiple Linux servers that run under multiple KVM systems on multiple LPARs on multiple LinuxONE servers. No SPoFs for the application remain.

We have eliminated the SPoFs by using a second LinuxONE server (System B) to host our second LPAR that will run KVM and Linux guests. The application is installed on Linux servers in each LPAR. During normal operations, each LPAR receives 50% of the workload of the application.

However, a hidden cost of this architecture is that both LinuxONE servers must be sized to run 100% of the workload should the other server fail. Software costs also increase because of this.

It is more cost-effective to run a System z LPAR near 100% CPU utilization. But the above architecture would run each LPAR nearer to 50% utilization, so that it has extra capacity in case of a failure of the other LPAR. There are several alternatives that allow you to correctly size the LPAR to run 50% of the workload and keep the utilization nearer to 100%:

1. Configure fewer CPUs than are needed to run 100% of the workload in the LPAR. Configure other CPUs as standby CPUs that can be brought online quickly with Capacity Upgrade on Demand. When extra capacity is needed:
   1. The standby CPUs are defined as “active” to the LPAR.
   2. KVM varies the new CPUs online.

This process is non-disruptive and can be automated or completed manually in a few minutes.
• Run other lower-priority work in each LPAR. Configure a KVM for IBM z Hypervisor Performance Manager (HPM) policy so that those running production workloads have a higher priority than those running “other” work. If a failover occurs, KVM will give the system’s CPU and memory to the production guests and withhold CPU and memory from the other workloads.

• Run lower-priority work (such as development or test) in a separate LPAR that shares its CPUs with the production LPAR. Run these LPARs on both System A and System B, so in case either fails there is low-priority work that can be interrupted. LPAR weights ensure that the production LPAR gets the majority of the CPUs.

• Use Capacity Backup (CBU) to bring added CPUs online when needed.

Summary
Examples 2 - 4, and all of the reference architectures illustrated in this paper use clusters containing two nodes. You can always choose to instead create a cluster of three servers. In our examples that use one server type per LPAR, you could instead define three LPARs. The advantage of a three-member cluster over a two-member cluster is that should one cluster member fail, you still retain a cluster of two members, and a good degree of high availability. With a two-member cluster, if one member fails then you are now running in non-HA mode on the single remaining member, until the failed cluster member can be brought back online.

When the absolute highest levels of availability must be maintained at all times, it is recommended to use a cluster of three LPARs. Otherwise a cluster of two LPARs is typically sufficient.

The above example, titled “Example 3: High Availability Needed,” shows the most cost-effective solution for architecting LPARs and KVM for High Availability. For such an implementation, the following is recommended:

• Use a single LinuxONE server so that your KVM LPARs can share the same CPUs.

• Use two to three LPARs to run your production workload.

• Create clusters of applications split between Linux servers running in each KVM LPAR.

• Run your test and development Linux virtual servers either in:
  o Their own LPAR. You can use the following LPAR weights as starting values:
    ▪ Production1: 35%
    ▪ Production2: 35%
    ▪ Test/Dev: 30%
  o One of the production LPARs, and give that LPAR more resources than the other production LPARs. You must ensure that the production guests have priority in the HPM policy for getting system resources.
Chapter 1: Scenarios
The reference architectures in this document address two typical customer scenarios. The scenarios build on each other, increasing in complexity.

For all scenarios, our goal is to provide a reference architecture that:

- Is rapidly scalable to support increases or decreases in business volume. Many times this can be accomplished simply by bringing more CPUs online to the existing architecture.
- Provides rapid failover.

Scenario: Active-Active KVM Cluster with Opaque VMs
You have a variety of critical applications that run on Linux on System z inside KVM virtual machines. The applications do not necessarily have any built-in HA features, but will restart whenever the virtual machine they are running in is rebooted. You require rapid restart of the virtual machines in which the applications run.

Scenario: Active-Active KVM Cluster with Guest Node VMs
You have a critical application that runs on Linux on System z inside a KVM virtual machine. To meet availability goals for the application, in addition to failing over the entire virtual machine you also require the ability to monitor and fail over individual services running inside the virtual machine.
Chapter 2: Reference Architecture: Active-Active KVM Cluster with Opaque VMs

Scenario Being Solved

You have a variety of critical applications that run on Linux on System z inside KVM virtual machines. The applications have no built-in HA features, and will restart whenever the virtual machine they are running in is rebooted. You require rapid restart or failover of the virtual machines in which the applications run.

Architecture Principles

This architecture is designed to follow these principles:

- Software is generally considered less reliable than hardware. The IBM LinuxONE server hardware contains redundant components, making its MTBF (Mean Time Between Failure) in the range of years. Because the hardware is so reliable, we allow the LinuxONE server to be a single point of failure in this architecture. We duplicate all the software environments (LPAR, KVM and Linux) so that none of them is a single point of failure.
- Outage time visible to the end user will be minimized, regardless of whether a virtual machine or an entire KVM host fails.
- Automated rebooting of a failed virtual machine is an acceptable recovery action.
- Maintaining data integrity for systems and applications is critical, even if doing so extends recovery time.
- The base architecture anticipates a low enough volume that it can be managed by a single server, but can scale if necessary to support increases and decreases in business volume.
In this architecture, Pacemaker and Corosync run on the KVM hosts, with each host considered a node in the cluster. The virtual machines (VMs) are treated as black boxes. Pacemaker starts and stops them, and knows what KVM (i.e., libvirt) knows about their status, but that is all – it is unaware of what applications are running in the VMs. All KVM nodes are active in the cluster, and the VMs are spread across them. This architecture should scale up to 16 KVM nodes (i.e., hypervisors) and virtually any number of VMs in the same cluster.

### Flow of requests through this architecture

1. **Corosync.** Provides the cluster communication layer, and establishes cluster membership and quorum. Dual networking rings are configured for redundancy, each on a different network segment. If one network ring fails, communications will transparently fail over to use the other ring. For 2-node clusters, a special Corosync configuration option (two_node:1) is set to enable it to enable it to manage quorum properly when one node is lost.

2. **Cluster Information Base (CIB).** Contains the availability policy defined by the user (through line mode commands using the “pcs” subshell, and/or through the pcsd Web UI), and tracks the configuration and status of all cluster resources. It is owned by the designated coordinator node, and dynamically replicated from there to all other nodes.

3. **Policy Engine (PE).** Maps the cluster’s current state to the desired state and initiates required actions by feeding them to the CRMd acting as the designated coordinator.
4. **Cluster Resource Manager (CRMd).** Receives instructions from the designated coordinator node and passes them along to the local resource manager.
   a. **Designated coordinator:** The CRMd on a node elected from all available nodes to act as the focal point for the cluster. It receives instructions from the policy engine and passes them along via Corosync to the CRMd on other nodes (and to the local resource manager on its own node).

5. **Local Resource Manager (LRMd).** Receives instructions from the CRMd and carries them out by driving actions through the appropriate resource agents.

6. **Guest XML and Image Files.** The KVM virtual machine domain definition XML files and qcow2 image files are housed on a network-accessible Linux LPAR, which is configured as an NFS server. It exports the directories containing those files (xml files as read-only, qcow2 files as read/write) so they are accessible to all cluster nodes. **NOTE:** in addition to image-based guests, disk-based guests may also be employed. They must reside on LUNs accessible to all KVM hosts in the cluster (not shown).

7. **Virtual Machines.** All virtual machines are managed by a VirtualDomain resource agent. Pacemaker starts, stops, and live migrates (optionally) virtual machines as needed. Any virtual machine is able to run on any KVM host in the cluster. In the diagram, active virtual machines are encased in solid lines; dotted lines around a virtual machine illustrate a virtual machine which is not active on that host, but is eligible to be moved there by Pacemaker if required.

8. **Resource Agents (RAs).** Can be many per node, each specific to a particular managed resource (a virtual machine, an Ethernet adapter, etc.). The RA translates generic cluster instructions passed to it by the LRMd (such as “restart the resource”) into commands specific to that particular resource type.
   a. **VirtualDomain RA.** A unique VirtualDomain RA is defined for each virtual machine in the cluster. Pacemaker has sole control over these VMs. They are only managed by Pacemaker, and are not auto-started by KVM when it boots, and are never started manually by a human or through any means other than via Pacemaker. The VMs are opaque to Pacemaker, managed like any other resource but with no visibility to services running inside each VM. If a VM fails, Pacemaker will restart it in place. If the restart in place fails, it will be retried up to the threshold of tries specified in the Pacemaker policy; if that threshold is exceeded within a time window specified in the policy, Pacemaker will attempt to restart the VM on another host in the cluster. If the entire KVM host fails, Pacemaker will restart all of the virtual machines that had been running on that failed node over on surviving nodes. If desired, Pacemaker can be configured to use live guest migration to dynamically move virtual machines around to other nodes in the cluster to balance workload, or to evacuate a node prior to a planned outage.
   b. **FileSystem RA (optional).** A unique FileSystem RA is cloned across all KVM nodes for each of the NFS exported directories (guest xml files and qcow2 image files), which mounts them on each KVM node. Colocation and ordering constraints are defined in the Pacemaker policy to ensure those file systems are mounted before their VirtualDomains are started.
   c. **MailTo RA (optional).** For a high-importance virtual machine, a MailTo RA can be defined and placed in a resource group with the VirtualDomain RA. If that virtual machine is started/stopped/moved, the MailTo RA will notify the system administrator via an email. Caution should be used here to limit the number of such MailTo RAs to avoid an email flood in the event of a KVM node outage.

9. **System z Hardware Management Console (HMC).** The HMC is configured to enable its REST API for use by the fence_ibmz STONITH agent with appropriate authentication.
10. **STONITH agent (fence_ibmz)**. If Pacemaker loses contact with a KVM node in the cluster, it will invoke the fence_ibmz STONITH (Shoot The Other Node In The Head) agent, which in turn will communicate with the System z HMC and tell it to shut down the LPAR in which that KVM node was running. This ensures that a node which Pacemaker BELIEVES is down, really IS down. To ensure data integrity, no resources from the failed node will be restarted elsewhere in the cluster until that fencing operation has completed. The fence_ibmz agent will periodically (every 60 seconds by default) invoke a monitor action to confirm it retains network connectivity to the System z HMC, ensuring it will be ready when needed. The fence_ibmz agent is eligible to run on either KVM host node.

   a. The Pacemaker policy can specify that when a node detects it has lost contact with the rest of the cluster and so no longer has quorum, it should take it upon itself to stop all managed resources running on it (i.e., a no quorum policy of “stop”). That will cause it to try and shut down all of its Pacemaker-managed virtual machines. Ideally, all will be shut down cleanly before the fencing action completes and deactivates the LPAR. However, there is no guarantee that will happen, so some virtual machines may be shut down hard, which could lead to the need for them to run fsck against their file system when they are subsequently rebooted.

**Product Versions**

- Pacemaker version 1.1.13 and Corosync version 2.3.4 or higher. These are delivered with KVM for IBM z Systems version 1.1.2.

**Planned Outages**

For a planned outage of a KVM node, all virtual machines can be evacuated from the node by using a single Pacemaker command to set the node in “standby” mode, triggering Pacemaker to live migrate the virtual machines away from the target node to other nodes in the cluster (or, if Pacemaker-driven live migration has not been enabled for the virtual machines, to shut them down and reboot them elsewhere).

For planned outages of individual virtual machines, the VM can be shut down in an orderly fashion from Pacemaker.
What We Learned in Testing

When this architecture was set up and tested for planned and unplanned outages, we learned the following:

- Did the software failover as expected? Yes.
- Did users experience any outage time or transactions that they needed to retry? Yes.
- Did users experience any permanent data loss? No.
- How long did the failover take? (How long did users experience outages):
  - VM failure: This depends on the interval configured in Pacemaker for checking the status of a virtual machine. In our case, when a VM crashed, the time it took for Pacemaker to detect the failure and restart the VM up to the point of being able to respond to a network ping was on average approximately 23 seconds.
  - Host failure: When an entire KVM host with 64 GB of memory crashed, the time to fence the failed node and restart its VMs on a surviving KVM node in our environment was approximately 410 seconds (for 16 VMs).
    - Note that the action taken by the fence_ibmz fencing agent varies depending on the state of the LPAR. If the LPAR is in not in an “operating” state (e.g., KVM crashed with a kernel panic), then the fencing agent will conclude that the LPAR is already down and so it takes no action. However, if the LPAR is in an “operating” state (e.g., it is hung or has lost network access with the rest of the cluster, but is still running), then the fencing agent will deactivate the LPAR.
    - Note also that in the case of the fencing agent deactivating the LPAR, the fencing time increases as the amount of memory assigned to the KVM LPAR increases (because memory is zeroed as part of the deactivate process). In the example timings shown here, the fencing agent did perform an LPAR deactivate.

Architectural Decisions

Architectural decisions were made based on the following key criteria:

- High Availability
- Cost
- Simplicity

<table>
<thead>
<tr>
<th>Architectural Decisions</th>
<th>Pros/Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decision Point</td>
<td></td>
</tr>
<tr>
<td>Use two separate System z servers to host the two LPARs.</td>
<td>This architecture can easily be run on two separate System z servers. All VM’s running on a KVM node can be simply failed over to an LPAR on another physical server.</td>
</tr>
<tr>
<td>Symmetric “opt-out” cluster vs asymmetric “opt-in” cluster</td>
<td>A cluster can either be configured as symmetric or asymmetric. A symmetric cluster is one where by default any resource can run on any node and Pacemaker will attempt to balance resources across nodes. If pure symmetry is not desired, location constraints</td>
</tr>
<tr>
<td>Corosync communication path redundancy</td>
<td>This architecture uses dual network rings (on different network segments) to provide redundancy for the communication path used by Corosync to maintain cluster membership and quorum. This straightforward approach can remove network adapters and network switches as single points of failure. Alternatively, redundant network adapters on each host could be bonded together using the Linux bonding driver and attached to a single Corosync ring. That approach is certainly viable, but by itself does not provide redundancy for network segments and their switches, so would require an additional approach at the network switch layer for protecting against link failure. For even more redundancy, the two approaches could be combined.</td>
</tr>
<tr>
<td>Resource stickiness</td>
<td>By default, Pacemaker assumes it can move resources around the cluster at will. For example, assume a KVM node is fenced and its VMs are restarted on surviving nodes. When the failed KVM node is rebooted and rejoins the cluster, Pacemaker will attempt to immediately move a subset of VMs over to that node to balance VMs around the cluster. If live migration is not enabled then this will cause those VMs to be shut down where they are running and rebooted on the newly-restarted KVM node. This creates an additional outage to those VMs and so is typically undesirable behavior. Even if live migration is being used, suddenly migrating many guests around the cluster in the middle of the day is probably still undesirable. To prevent this, the default can be changed to assign a non-zero “stickiness” value to all resources, which tells Pacemaker it should prefer to keep resources running where they are if possible.</td>
</tr>
<tr>
<td>FileSystem Resource Agent</td>
<td>Use of the FileSystem resource agent to mount the NFS exported guest xml files and qcow2 image files is optional. It is used in this architecture in conjunction with colocation and ordering constraints in the Pacemaker policy to ensure that Pacemaker does not needlessly attempt to start 100’s of virtual machines if an NFS mounting error has occurred and their critical files are not even available.</td>
</tr>
<tr>
<td>Active-Active vs Active-Passive</td>
<td>This architecture depicts an Active-Active scenario in which all KVM nodes are actively running virtual machines managed by Pacemaker (though obviously a particular virtual machine is only running in one KVM node at a time). By making use of location constraints in the Pacemaker policy it is also possible to deploy an Active-Passive architecture in which by default all VMs run on one KVM node, and another KVM node is up in the cluster but not running any VMs. Should the active node fail, all its VMs would be shifted over to the surviving node. Such a configuration could be used in conjunction with System z capabilities such as Capacity Upgrade on Demand to...</td>
</tr>
<tr>
<td>Key problem areas</td>
<td></td>
</tr>
<tr>
<td>-------------------</td>
<td></td>
</tr>
<tr>
<td>If only a single Corosync communication ring exists in the cluster and access to it fails permanently from one or more nodes, Pacemaker will treat that the same as a failure of those nodes and manage failover appropriately. However, if multiple nodes concurrently experience a transient network failure that temporarily interrupts communication over that ring which is then re-established before the nodes can be fenced, Pacemaker may get confused and results are unpredictable. Maintaining redundant corosync communication rings on different network segments provides protection against such a failure.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Implementation Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Some configuration on the System z Hardware Management Console (HMC) is required to enable the fence_ibmz STONITH agent to interact with it. Refer to the book <em>KVM for IBM z Systems v1.1.2: System Administration</em> for guidance on how to accomplish this (see the References section of this paper for a link).</td>
</tr>
<tr>
<td>When doing initial Pacemaker setup in a test configuration, you may find it helpful to set SELinux into permissive mode -- however, permissive mode is not recommended for a production system. If you choose to do this, then once the setup is complete and working you can make any necessary SELinux policy adjustments and put SELinux back into enforcing mode. The book <em>KVM for IBM z Systems v1.1.2: System Administration</em> provides guidance on making such SELinux policy adjustments (see the References section of this paper for a link).</td>
</tr>
</tbody>
</table>
• For symmetric clusters (see “Architectural Decisions”), be aware that virtual machines enabled for live guest migration but without stickiness attribute values may be moved multiple times as guests are balanced by Pacemaker among the surviving cluster nodes.

• The supported limit for the size of a Corosync cluster is 16 nodes, and this architecture should scale to that size. However, testing in support of this paper was limited to five nodes.
Chapter 3: Reference Architecture: Active-Active KVM Cluster with Guest Node VMs

Scenario Being Solved
You have a variety of critical applications that run on Linux on System z inside KVM virtual machines. The applications do not necessarily have any built-in HA features, but will restart whenever the virtual machine they are running in is rebooted. You require rapid restart or failover of the virtual machines in which the applications run, and also for certain resources running within the virtual machines.

Architecture Principles
This architecture is designed to follow these principles:

- Software is generally considered less reliable than hardware. The LinuxONE server hardware contains redundant components, making its MTBF (Mean Time Between Failure) in the range of years. Because the LinuxONE server hardware is so reliable, we allow the LinuxONE server to be a single point of failure in this architecture. We duplicate all the software environments (LPAR, KVM, and Linux) so that none of them is a single point of failure.
- Outage time visible to the end user will be minimized, regardless of whether a managed resource, a virtual machine or an entire KVM host fails.
- Automated rebooting or failover of a failed virtual machine is an acceptable recovery action if the entire virtual machine fails. Automated restarting of failed resources that run within a virtual machine is also desired, even if the resource is failed over to a different virtual machine.
- Maintaining data integrity for systems and applications is critical, even if doing so extends recovery time.
- The base architecture anticipates a low enough volume that it can be managed by a single server, but can scale if necessary to support increases and decreases in business volume.
In this architecture, Pacemaker and Corosync run on the KVM hosts, with each host considered a node in the cluster. Each virtual machine (VM) also runs the lightweight pacemaker-remote agent, which turns each VM into a managed “guest node” in the cluster. These VMs are treated as cluster nodes just like the KVM hosts, except they do not participate in quorum voting and cannot initiate a STONITH fencing action. Pacemaker running on the KVM host is able to start, stop, and monitor cluster resources inside of each guest node, as well as on the KVM nodes. All KVM nodes are active in the cluster, and the VMs are spread across them. This architecture should scale up to 16 KVM nodes and virtually any number of guest node VMs in the same cluster.

The example scenario uses an HTTP server with a floating Service IP as the example application. A Service IP address is a single IP address by which the HTTP server is known and provides service to the outside world. In the event of a failover that leads to a standby instance of the HTTP server being started on another VM, this IP address must be moved over to the new VM. This choice offers the following benefits as an example:

- An HTTP server is a common application for Linux, yet is lightweight enough that another application could be easily substituted.
- The use of a Service IP not only illustrates solid availability, it also represents an additional required resource for the example. Even the simplest of Web-serving arrangements requires at least one additional resource for dynamic content. Demonstrating the failover considerations afforded by a Service IP will provide a multi-resource example that can generalize to other examples requiring multiple resources.
Flow of requests through this architecture
1. **Corosync.** Same as the prior architecture.
2. **Cluster Information Base (CIB).** Same as the prior architecture.
3. **Policy Engine (PE).** Same as the prior architecture.
4. **Cluster Resource Manager (CRMd).** Same as the prior architecture
5. **Local Resource Manager (LRMd).** Same as the prior architecture.
6. **Guest XML and Image Files (not shown).** Same as the prior architecture.
7. **Virtual machines.** All virtual machines are managed by a VirtualDomain resource agent. Pacemaker starts, stops, and live migrates (optionally) virtual machines as needed. Any virtual machine is able to run on any KVM host in the cluster.
   a. The VMs run the **pacemaker-remote** agent and so are treated as “guest nodes” by Pacemaker on the KVM host, giving it full visibility to, and control over, managed services running inside each VM.
8. **Resource Agents (RAs).** Can be many per node, each specific to a particular managed resource (a virtual machine, an Ethernet adapter, etc.). The RA translates generic cluster instructions passed to it by the LRMD (such as “restart the resource”) into commands specific to that particular resource type.
   a. **VirtualDomain RA.** A unique VirtualDomain RA is defined for each virtual machine in the cluster. Pacemaker has sole control over these VMs. They are only managed by Pacemaker, and are not auto-started by KVM when it boots, and are never started manually by a human or through any means other than via Pacemaker. If a VM fails, Pacemaker will restart it in place. If the restart in place fails, it will be retried up to the threshold of tries specified in the Pacemaker policy; if that threshold is exceeded within a time window specified in the policy, Pacemaker will attempt to restart the VM on another host in the cluster. If the entire KVM host fails, Pacemaker will restart all of the virtual machines that had been running on that failed node over on surviving nodes. If desired, Pacemaker can also be configured to use live guest migration to dynamically move virtual machines around to other nodes in the cluster to balance workload, or to evacuate a node prior to a planned outage.
      i. The virtual machine will run a pacemaker-remote agent, which will interact with Pacemaker on the host. An implicit internal resource for Pacemaker’s **remote connection** to the guest is created as a result of specifying the meta attribute “remote-node” on the VirtualDomain resource definition. This implicit remote connection resource is shown in the diagram with a dotted line and gray text, though in reality it is internal to the CRMd component. The remote-node meta attribute specifies the actual host name of the VM, which must be resolvable to the IP address of the guest node (e.g., via a DNS or /etc/hosts). Note that this is the address of the VM itself, which is different from the floating Service IP address that is associated with the HTTP server application in this example.
   b. **IPaddr2 RA.** Defines a floating Service IP address, which in this example will be associated with the HTTP server. It is defined as a second “aliased” IP address on top of a virtual network interface in the guest. Should the guest node fail, this IPaddr2 resource will be moved over to a surviving eligible (based on any location constraints) guest node and will in turn activate the service IP address by re-aliasing it on top of a network interface on that virtual machine. All
eligible guest nodes must have identical HTTP server configurations, so that no matter where the IPAddr2 resource is moved among nodes in that eligible pool, the Apache resource agent will later be able to start the Apache HTTP server successfully.

c. **Apache RA.** Starts, stops, and monitors an Apache HTTP server. The RA’s monitor operation will (by default) periodically load the server status page to ensure the HTTP server is still active. Should the HTTP server fail, the apache resource agent will restart it in place. Should the guest node the HTTP server is running within fail, the resource will be moved to a surviving guest node where the IPaddr2 resource has already been moved and restarted. There the resource agent will start a standby instance of the HTTP server.

d. **FileSystem RA (optional).** Same as the prior architecture (the NFS-mounted guest image and domain XML files are omitted in the diagram for simplicity).

9. **Colocation, ordering and location constraints.** A colocation constraint is defined in the Pacemaker policy to ensure the apache and IPAddr2 resources are always co-located on the same node. An ordering constraint ensures the IPaddr2 resource is started before the apache resource (to ensure the Apache HTTP server can bind to the Service IP address correctly at start up).

   a. If this is a symmetric cluster (see “Architectural Decisions” below), a location constraint (not shown) is defined in the Pacemaker policy to ensure that these resources are only started on guest nodes (as opposed to KVM nodes) – this is required because in a symmetric cluster, as far as Pacemaker is concerned all nodes (host and guest nodes) are equally eligible to run these resources unless Pacemaker is told otherwise.

   b. If this is an asymmetric cluster (see “Architectural Decisions” below), a location constraint (not shown) defines specifically which guest nodes the apache and IPaddr2 resources are allowed to run on (for example, if the HTTP server is only configured on a subset of nodes in the cluster).

10. **System z Hardware Management Console (HMC).** Same as the prior architecture.

11. **STONITH agent (fence.ibmz).** Same as the prior architecture. Note that only the KVM host nodes running the full Pacemaker stack, not the guest nodes, can initiate a STONITH action.

12. The Pacemaker policy can specify that when a node detects it has lost contact with the rest of the cluster and so no longer has quorum, it should take it upon itself to stop all managed resources running on it (i.e., a no quorum policy of “stop”). That will cause it to try and shut down all of its Pacemaker-managed virtual machines. Ideally, all will be shut down cleanly before the fencing action completes and deactivates the LPAR. However, there is no guarantee that will happen, so some virtual machines may be shut down hard, which could lead to the need to run fsck when they subsequently are rebooted.

**Product Versions**
- Pacemaker version 1.1.13 and Corosync version 2.3.4 or higher. These are delivered with KVM for IBM z Systems version 1.1.2.

**Planned Outages**
For a planned outage of a KVM or guest node, all managed resources can be evacuated from the node by using a single Pacemaker command to set the node in “standby” mode, triggering Pacemaker to move the resources
away from the target node to other nodes in the cluster. Once a guest node has been evacuated, the VM can be shut down in an orderly fashion from Pacemaker.
What We Learned in Testing

When this architecture was set up and tested for planned and unplanned outages, we learned the following:

- Did the software failover as expected? Yes.
- Did users experience any outage time or transactions that they needed to retry? Yes
- Did users experience any permanent data loss? No
- How long did the failover take? (How long did users experience outages):
  - VM failure: This depends on the interval configured in Pacemaker for checking the status of a virtual machine. In our case, when a VM crashed, the time it took for Pacemaker to detect the failure and restart the VM up to the point of being able to respond to a network ping was on average approximately 36 seconds.
  - Host failure: When an entire KVM host with 64 GB of memory hung, the time to fence the failed node and restart its VMs on a surviving KVM node in our environment was approximately 400 seconds (for 16 VMs).

  - Note that the action taken by the fence_ibmz fencing agent varies depending on the state of the LPAR. If the LPAR is in not in an “operating” state (e.g., KVM crashed with a kernel panic), then the fencing agent will conclude that the LPAR is already down and so it takes no action. However, if the LPAR is in an “operating” state (e.g., it is hung or has lost network access with the rest of the cluster, but is still running), then the fencing agent will deactivate the LPAR.

  - Note also that in the case of the fencing agent deactivating the LPAR, the fencing time increases as the amount of memory assigned to the KVM LPAR increases (because memory is zeroed as part of the deactivate process). In the example timings shown here, the fencing agent did perform an LPAR deactivate.

Architectural Decisions

Architectural decisions were made based on the following key criteria:

- High Availability
- Cost
- Simplicity

<table>
<thead>
<tr>
<th>Architectural Decisions</th>
<th>Pros/Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decision Point</td>
<td></td>
</tr>
<tr>
<td>Pacemaker remote vs full Pacemaker stack in each guest</td>
<td>Rather than using the Pacemaker-remote approach, the entire Pacemaker/Corosync stack could be installed in each guest, creating a pure VM cluster with no KVM host participation. There are a few downsides to that approach: 1) each guest must endure the overhead of configuring and running the full stack rather than just the lightweight pacemaker-remote agent; 2) redundant Corosync rings would need to be configured between all guests to ensure availability and integrity of the cluster can be maintained; 3) the number of guests in a single cluster would be limited by the practical Corosync limit of 16 nodes (when using Pacemaker remote, there is no defined limit for how many guest nodes can participate in a single cluster). Leveraging pacemaker-remote to</td>
</tr>
</tbody>
</table>
create guest nodes in the cluster is lighter weight and more flexible, so was depicted in this architecture.

On the other hand, when using pacemaker-remote the guest nodes cannot participate in quorum voting and cannot initiate fencing actions. When running the full Pacemaker stack on each guest, there is an external/libvirt STONITH agent that can be used by a guest to contact the host KVM and tell it to kill a failing VM – however, that agent has a limitation that it can only interact with a single hypervisor, so you cannot spread the VMs in the cluster across multiple KVM hosts, making the KVM host a single point of failure. Nonetheless, if quorum voting and initiating fencing actions are important for a particular solution, then it would become necessary to deploy the full stack in each guest.

<table>
<thead>
<tr>
<th>Colocation and ordering constraints vs resource groups</th>
<th>The colocation and ordering constraints shown in this architecture could instead be represented in the Pacemaker policy through a resource group. Since a group was shown in the prior architecture, the individual constraints are shown here simply to demonstrate a different approach.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use two separate System z servers to host the two LPARs.</td>
<td>Same as the prior architecture.</td>
</tr>
</tbody>
</table>
| Symmetric “opt-out” cluster vs asymmetric “opt-in” cluster | A cluster can either be configured as symmetric or asymmetric. A symmetric cluster is one where by default any resource can run on any node and Pacemaker will attempt to balance resources across nodes. If pure symmetry is not desired, location constraints can be set in the Pacemaker policy to prevent any given resource from running on a specific subset of nodes, creating an “opt-out” mechanism. The use of location constraints becomes especially important when using pacemaker-remote guest nodes to differentiate which resources should run on guest nodes and which should run on KVM host nodes (see “Implementation Notes” below).

An asymmetric cluster is one in which by default no resource is allowed to run on any node; location constraints in the Pacemaker policy must be set to specifically tell Pacemaker which nodes a given resource is allowed to run – creating an “opt-in” cluster.

This architecture will work with either type of cluster. Which to choose is simply a matter of which approach is more convenient in a given environment. |
| Corosync communication path redundancy | Same as the prior architecture.                                                                                                                                 |
| Resource stickiness | Same as the prior architecture.                                                                                                                                  |
| FileSystem RA | Same as the prior architecture.                                                                                                                                  |
| Active-Active vs Active-Passive | Same as the prior architecture.                                                                                                                                  |
Key problem areas

- Same as the prior architecture

Implementation Notes

- As noted earlier, Pacemaker treats KVM host nodes and pacemaker-remote nodes as equally eligible to run any resource. One implication of this is that if you are running a symmetric cluster that includes both ordinary VirtualDomain resources, and VirtualDomain resources that operate as pacemaker-remote guest nodes, Pacemaker may try to start ordinary VirtualDomain resources (or other host resources) inside of a pacemaker-remote guest node (i.e., a VM inside of a VM), which will fail. To prevent this, some sort of location constraint is required which tells each ordinary VirtualDomain resource not to attempt to start on a pacemaker-remote guest node (Pacemaker is smart enough not to attempt to start a pacemaker-remote guest node on another pacemaker-remote guest node). The best way we found to accomplish this is through the use of a node attribute called "#kind" that is established as soon as the node is created. This attribute can have one of three values: 1) "cluster" (normal node); 2) "remote" (ocf:pacemaker:remote node), or "container" (pacemaker remote guest node). So, for example, a pcs command to create a location constraint to prevent VirtualDomain resource “VM1_res” from ever being started on a guest node would look like this:
  o pcs constraint location VML_res rule score=-INFINITY "#kind" eq container

- Also note that the pacemaker-remote guests will need to be able to communicate over the corosync ring networks.
- Other implementation notes are the same as the prior architecture
Appendix 1: References

- Corosync home page: http://corosync.github.io/corosync/
- Pacemaker home page: http://clusterlabs.org/
- OCF resource agents from heartbeat: http://www.linux-ha.org/wiki/Resource_agents
- **VirtualDomain** heartbeat resource agent info: http://www.linux-ha.org/wiki/VirtualDomain_%28resource_agent%29

Comments on this paper.

Please address comments to:

Scott Loveland, d10swl1@us.ibm.com.